Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Modeling a Landing Gear System in Event-B

Amel Mammar and Régine Laleau

Institut Mines-Télécom/Télécom SudParis, CNRS UMR 5157 SAMOVAR, France
Université Paris-Est, LACL, UPEC, IUT Sénart Fontainebleau, France

Received: date / Revised version: date

Abstract. This article describes the Event-B modeling of a landing gear system of an aircraft whose the complete description can be found in [4]. This real-life case study has been proposed by the ABZ'2014 track that took place in Toulouse, the European capital of the aeronautic industry. Our modeling is based on the Parnas and Madey's 4-Variable Model that permits to consider the different parts of a system. These parts are incrementally introduced using the Event-B refinement technique. The entire development has been carried out with the Rodin toolset. To ensure the correctness of the different components, we use several verification techniques (animation, model-checking and proof) depending on the complexity and the kind of the properties to verify. Basically, prior to the proof phase that can be tedious and complex, we use the animator AnimB and the model checker ProB that permit to discover some trivial inconsistencies. Once no error is reported, we start the proof phase by using the Atelier B and SMT provers which we installed under Rodin. We conclude the article by drawing up some key findings of and lessons learned from this experience.

1 Introduction

Nowadays, the use of formal methods to ensure the correctness of critical complex systems is well-accepted. Indeed, any failure in the development of such systems may lead to loss of lives. The current paper reports on our experience in modeling and verifying a landing gear system of an aircraft, a case study proposed by the ABZ'2014 track, using Event-B [3] and its associated tools. The Event-B method permits to master the complexity of the system to develop thanks to the stepwise refinement that allows to gradually introduce the different parts of the system starting from an abstract model to a more concrete one. A stepwise refinement approach produces a correct specification by construction since we prove at each step the different properties of the system. A detailed description of the Event-B method is provided in Section 3.

The objective of the landing gear system is to permit a safe extension/retraction of the gear when the plane is going to land/fly. Each gear is placed in a landing-gear box equipped with a door that must be open when a gear is extending/retracting and closed when it becomes completely extended/retracted and locked. To this aim, the controller (See Figure 1) reads, periodically through a set of sensors, the states of the different elements (door, gear, handle, etc.) and sends orders to a set of electrovalves that make, for instance, the gear extend/retract or the doors open/close. More details will be introduced throughout the modeling of this system.
The system can be seen as continuously executing the following sequence of actions:

Do
Read Inputs from some sensors
Process Inputs to produce outputs
Send Outputs to the Electro-Valves
Until *a failure is detected*

The paper is organized as follows. Section 2 introduces the methodology we followed to model the landing gear system with the Event-B formal method. This formal method and the `ProB` model-checker are briefly described in Section 3. Sections 4-8 present the Event-B specifications that cover the different aspects of the system. The verification of the development and requirements are provided in Section 9. Section 10 draws up

Send offprint requests to: Amel Mammar, E-mail: amel.mammar@telecom-SudParis.eu

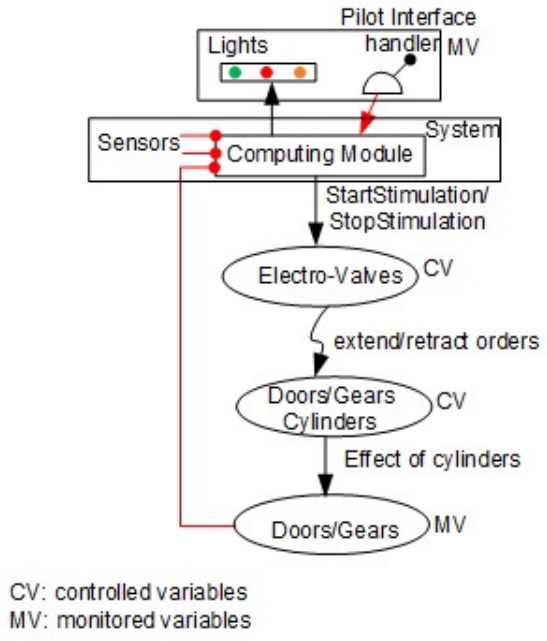


Fig. 1. The overall structure of the landing gear system

some key findings of and lessons learned from this experience.

2 Methodology

When modeling critical complex systems, we have to face up to the following main problems:

1. What are the different components constituting them, the environment with which they interact by exchanging information and finally the borders between them?
2. What is the starting-point of the modeling, that is, which element of the system or its environment has to be modeled at first?
3. At which moment, the degraded mode should be considered?

To answer the first problem, we were inspired by the four-variable model of Parnas and Madey [18]. The model distinguishes two groups of variables *environment* and *controller* variables (See Figure 2 taken from [18]):

1. *Environment variables*: represent the status of the elements outside the controller. Two kinds of variables are distinguished:
   - *Monitored variables*: the values of these variables are not calculated by the controller (represented by *software* in Figure 2) but can be monitored using input devices.
   - *Controlled variables*: the values of these variables are determined by the controller then transmitted through the output devices.
2. *Controller variables*: the values inside the controller system. Mainly, they represent the values of some elements as seen by the controller but also the different orders it sends.
   - *Input data*: the values stored in the controller and provided by some input devices like sensors.
   - *Output data*: they denote the orders sent by the controller toward the different environmental elements through output devices.

By juxtaposing Figures 1 and 2, we can deduce that:

- The *Software* and the *Input Devices* parts correspond to the *Controller* and the *Sensors*.
- The *Monitored variables* part corresponds to the physical components: *Door, Gear, Cylinders, etc.*.
- The *Controlled variables* part corresponds to the *Electro-Valves/Lights*.
- the *Input data* part corresponds to data provided by different sensors.
- Finally, the *Output data* part corresponds to data sent by the controller to the different kinds of *Valves*.

To answer the second problem about the elements to model at first, we have used the refinement mechanism proposed by some formal methods, such as Event-B for example. In such methods, we start by a very abstract model that represents the main functionality (goal) of the system, then details are gradually introduced thanks to the refinement technique. These details explain how the different elements of the system work together to achieve the main functionality of the system.

The main functionality of the landing gear system is to extend/retract the gear after opening the doors. Consequently, we start by describing the behavior of the gear, then the behavior of the doors that can be open/closed and the different monitored variables. Afterwards, we model the controlled variables that act on the monitored ones, mainly the different cylinders. Finally, we consider the input/output data that permit to link monitored/controlled variables to the system as follows:

$$Monitored\ variables \leftrightarrow Input\ data$$

$$Controlled\ variables \leftrightarrow Output\ data$$

To answer the last problem about the degraded mode, we have chosen to proceed in two steps [15]. In a first step, we consider that the system behaves correctly without any failure. Then, we introduce the failures in a second phase.

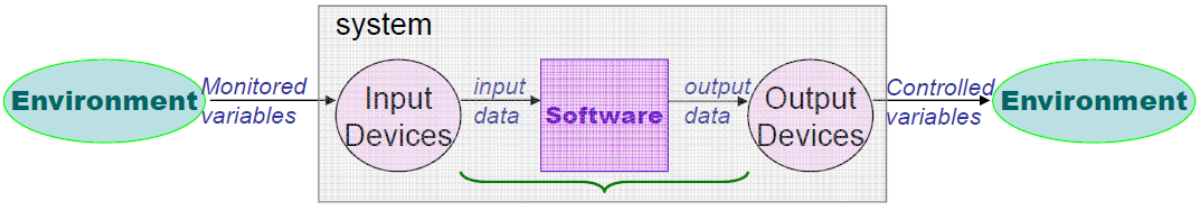


Fig. 2. The four-variable model

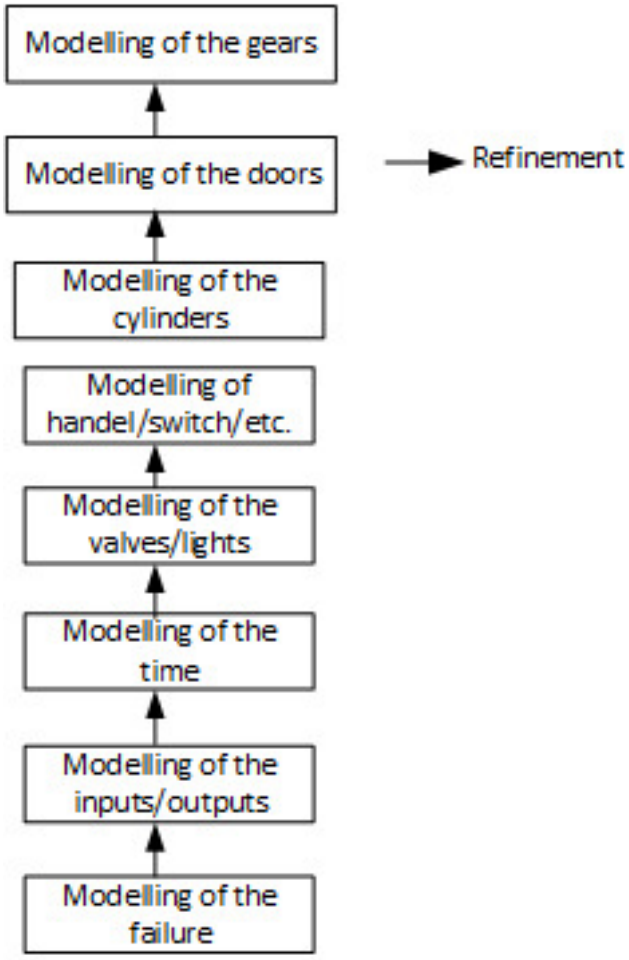


Fig. 3. Gears system design by refinement

From a practical point of view, the modeling is achieved in six main steps (See Figure 3):

1. *Modeling the monitored variables*: we describe the behavior of the physical components like the doors, the gear, the cylinders, but also the handle, the switch and the shock absorbers. The variables modeling these components are suffixed with "_*p*" to represent their actual (physical) status (See Section 4).
2. *Modeling the controlled variables*: we describe in this phase the behavior of the valves that permit to act directly on the doors, the gear and the cylinders (See Section 5). We also describe the behavior of the lights that inform the pilot about the status of the system in general. Again, the variables modeling these components are suffixed with "_*p*" since they represent their actual status (See Section 5).
3. *Modeling the input/output variables and the controller*: we describe how the controller reads information from the sensors, sends orders to the valves and how it updates the values of the lights (See Section 6).
4. *Modeling timing aspects*: to facilitate the design, we have chosen to elaborate a first modeling of the system without considering any timed constraints. The timed aspects are taken into account later by refinement (See Section 7).
5. *Modeling the failure cases*: in this step, we take into account the system anomalies caused by failures on the different elements of the system (See Section 8).
6. Finally, we describe how properties are verified (See Section 9.3).

In each of the previous steps, the different elements are gradually introduced thanks to the Event-B refinement mechanism. The next section gives a brief description of the Event-B method together with its refinement technique.

3 Overview of the Event-B method and the ProB model-checker

3.1 Event-B method

Event-B is the successor of the B method [2] permitting to model discrete systems using mathematical notations. The complexity of a system is mastered thanks to the refinement concept that allows to gradually introduce the different parts that constitute the system starting from an abstract model to a more concrete one. An Event-B specification is made of two elements: *context* and *machine*. A context describes the static part of an Event-B specification; it consists of constants and sets (user-defined types) together with axioms that specify their properties:

CONTEXT *Cont*
Sets *S*
Constants C
Axioms *A*
END

The dynamic part of an Event-B specification is included in a machine that defines variables V and a set of events E. The possible values that the variables hold are restricted using an invariant, denoted Inv, written using a first-order predicate on the state variables:

MACHINE *Name*
SEES *Cont*
Variables *V*
Invariants *Inv*
Events *E*

Each event has the following form:

ANY *X*
WHEN G
THEN *Act*
END

This event can be executed if it is enabled, i.e. all the conditions G, named guards, prior to its execution hold. Among all enabled events, only one is executed. In this case, substitutions Act, called actions, are applied over variables. In this paper, we restrict ourselves to the *becomes equal* substitution, denoted by $(x := e)$.

The execution of each event should maintain the invariant. To this aim, proof obligations are generated. For each event, we have to establish that:

$$\forall S, C, X.\ (A \wedge G \wedge Inv \Rightarrow [Act]Inv)$$

where $[Act]Inv$ gives the weakest constraint on the *before* state such that the execution of Act leads to an *after* state satisfying Inv.

Refinement is a process of enriching or modifying a model in order to augment the functionality being modeled, or/and explain how some purposes are achieved. Both Event-B elements *context* and *machine* can be refined. A context can be extended by defining new sets S_r and/or constants C_r together with new axioms A_r. A machine is refined by adding new variables and/or replacing existing variables by new ones V_r that are typed with an additional invariant Inv_r. New events can also be introduced to implicitly refine a **skip** event. In this paper, the refined events have the same form:

ANY X_r
WHEN G_r
THEN Act_r
END

To prove that a refinement is correct, we have to establish the following two proof obligations:

- *guard refinement*: the guard of the refined event should be stronger than the guard of the abstract one:

$$\forall(S, C, S_r, C_r, V, V_r, X, X_r).\ (A \wedge A_r \wedge Inv \wedge Inv_r \Rightarrow (G_r \Rightarrow G))$$

- *Simulation*: the effect of the refined action should be stronger than the effect of the abstract one:

$$\forall(S, C, S_r, C_r, V, V_r, X, X_r).\ (A \wedge A_r \wedge Inv \wedge Inv_r \wedge [Act_r]Inv_r \Rightarrow [Act]Inv)$$

To discharge the different proof obligations, the Rodin[1] platform offers an automatic prover but also the possibility to plug additional external provers like the SMT and Atelier B provers that we use in this work. Both provers offer an automatic and an interactive options to discharge the proof obligations.

3.2 The ProB model checker

ProB [17] is an animator and explicit automatic model checker, originally developed for the verification and validation of software development based on the B language. Developed at the University of Düsseldorf starting from 2003, ProB[2] implements an automatic model checking technique to check LTL (linear temporal logic) [20] and CTL (Computational Tree Logic) [9] properties against a B specification. The core of ProB is written in a logical programming language called Prolog; its purpose is to be a comprehensive tool in the area of formal verification methods. Its main functionalities can be summarized up as follow:

1. ProB can find a sequence of operations that, starting from a valid initial state of the machine, moves the machine into a state that violates its invariant,

[1] `http://www.event-b.org/install.html`
[2] https://github.com/bendisposto/probparsers

2. giving a valid state, ProB can exhibit the operation that make the invariant violated,
3. ProB allows the animation of the B/EventB specification to permit the user play different scenarios from a given starting state that satisfies the invariant. Through a graphical user interface implemented in Tcl/Tk, the animator provides the user with: (i) the current state, (2) the history of the operation executions that has led to the current state and (3) a list of all the enabled operations, along with proper argument instantiations. In this way, the user does not have to guess the right values for the operation arguments.
4. ProB supports the model checking of the LTL and CTL assertions.

4 Modeling the monitored variables

In the system, we have the following monitored variables: *gear, doors, cylinders, handle, hydraulic circuit* and *switch*. These elements are introduced according to the following refinement strategy:

- Initial model (Component *Gears*): we start by describing the behavior of the gear, that can be made extended or not, since this is the main objective of the system.
- 1st refinement (Component *GearsIntermediate*): as the extension/retraction of a gear can take time, we refine the state where a gear is not extended by distinguishing two different sub-states: retracted or partly extended.
- 2nd and 3rd refinements (Components *Doors* and *DoorsIntermediate*): like for the gear, we describe the state of a door as open or not, then we add an intermediate state to model a partly-open door.
- 4th refinement (Component *Cylinders*): in this step, we introduce the cylinders that allow the motion of the doors and gear.
- 5th refinement (Component *HandleSwitchShockAbsorber*): we model in this phase the handle, the analogical switch, the hydraulic circuit and the shock absorbers.

In the following sub-sections, we detail each step. The complete Event-B development is available at [1].

4.1 Gear modeling: the initial model and the first refinement (machines Gears and GearsIntermediate)

We first introduce a context *PositionsDoorsGears* with set `PositionsDG` representing the three possible cases for gear/door/etc.: *front*, *left* or *right*.

CONTEXT
PositionsDoorsGears
SETS
PositionsDG
CONSTANTS
front
right
left
AXIOMS
partition(PositionsDG, {*front*}, {*right*}, {*left*})

Then, we define a Boolean variable *gear_ext_p* to formalize whether a gear is extended or not. An other alternative is to use an enumeration with two values (*extended* and *notExtrended*, but we have chosen to use a Boolean variable because, at the first level, we want to model only two states (*extended* or not); these states are then refined by exhibiting two additional intermediate states (*retracted* and *moving*). However in EventB, it not possible to extend a set by adding new values.

VARIABLES *gear_ext_p*
INVARIANT
inv1: $gear\_ext\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$

To make the gear extended or not, we define the following two events:

`Make_GearExtended`
ANY *po* **WHERE**
$po \in \texttt{PositionsDG} \wedge$
$gear\_ext\_p(po) = \texttt{FALSE}$
THEN
$gear\_ext\_p(po) := \texttt{TRUE}$
END

and

`Start_GearRetract`
ANY *po* **WHERE**
$po \in \texttt{PositionsDG} \wedge$
$gear\_ext\_p(po) = \texttt{TRUE}$
THEN
$gear\_ext\_p(po) := \texttt{FALSE}$
END

When a gear is not extended, it can be retracted or partly-extended. So, we refine the previous specification by introducing a new Boolean variable *gear_ret_p* that is true if the gear is entirely retracted. This variable is defined by two invariants (**inv2**) and (**inv3**), where (**inv3**)

states that a gear cannot be extended and retracted at the same time:

VARIABLES *gear_ret_p*
INVARIANT
inv2: $gear\_ret\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv3: $\forall po.(po \in \texttt{PositionsDG} \Rightarrow \neg(gear\_ext\_p(po) = \texttt{TRUE} \wedge gear\_ret\_p(po) = \texttt{TRUE}))$

Consequently, the event `Make_GearExtended` is refined by adding the guard ($gear\_ret\_p(po) = \texttt{FALSE}$), and we define the two following new events to make a gear start extending (it becomes no longer retracted) or complete its closing.

`Start_GearExtend`
ANY *po* **WHERE**
$po \in \texttt{PositionsDG}$
$gear\_ret\_p(po) = \texttt{TRUE}$
THEN
$gear\_ret\_p(po) := \texttt{FALSE}$
END

and

`Make_GearRetracted`
ANY *po* **WHERE**
$po \in \texttt{PositionsDG}$
$gear\_ext\_p(po) = \texttt{FALSE}$
$gear\_ret\_p(po) = \texttt{FALSE}$
THEN
$gear\_ret\_p(po) := \texttt{TRUE}$
END

4.2 Doors modeling: the second and third refinements (machines Doors and DoorsIntermediate)

In this part, we present the modeling of the doors. To this aim, we have proceeded like for the gear by defining two levels. In the first level, we define a new variable *door_open_p* to know if a door is open or not. Then, we refine, in the second level, the state where a door is not open by adding a new variable *door_closed_p* to state if the door is closed or partly-open.

– the second refinement: at this level, we define the variable *door_open_p* and express an invariant to state that when a gear is partly-extended then all the doors are open. In other words, it is not possible to start the extending/retracting of a gear until all the doors are open.

VARIABLES *door_open_p*
INVARIANT
inv4: $door\_open\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv5:
$\exists po.(po \in \texttt{PositionsDG} \wedge gear\_ext\_p(po) = \texttt{FALSE} \wedge gear\_ret\_p(po) = \texttt{FALSE}) \Rightarrow door\_open\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\}$

In order to preserve the invariant (**inv5**), the events `Start_GearExtend` and `Start_GearRetract` are refined by adding the guard:

$$door\_open\_p = \text{PositionsDG} \times \{\texttt{TRUE}\}$$

In addition, we define two events to make a door open and start closing.

`Make_DoorOpen`
ANY *po* **WHERE**
$po \in \texttt{PositionsDG}$
$door\_open\_p(po) = \texttt{FALSE}$
THEN
$door\_open\_p(po) := \texttt{TRUE}$
END

and

`Start_DoorClose`
ANY *po* **WHERE**
$po \in \texttt{PositionsDG}$
$door\_open\_p(po) = \texttt{TRUE}$
$(gear\_ext\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\} \vee gear\_ret\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\})$
THEN
$door\_open\_p(po) := \texttt{FALSE}$
END

– the third refinement: in this level, we define the variable *door_closed_p* and express that a door cannot be open and closed at the same time:

VARIABLES *door_closed_p*
INVARIANT
inv6: $door\_closed\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv7: $\forall po.(po \in \texttt{PositionsDG} \Rightarrow \neg(door\_open\_p(po) = \texttt{TRUE} \wedge$
new va $door\_closed\_p(po) = \texttt{TRUE})$

In order to preserve invariant (**inv7**), we refine the event `Make_DoorOpen` by adding the guard:

$$door\_closed\_p(po) = \texttt{FALSE}$$

We also define two new events to make a door start opening (it becomes no longer closed) or accomplish its closing.

```
Start_DoorOpen
  ANY po WHERE
    door_closed_p(po) =TRUE
  THEN
    door_closed_p(po):=FALSE
  END
```

and

```
Make_DoorClosed
  ANY po WHERE
    door_closed_p(po) =FALSE
    door_open_p(po) =FALSE
  THEN
    door_closed_p(po):=TRUE
  END
```

4.3 Cylinders modeling: the fourth refinement (machine Cylinders)

The motion of the gear and the doors is performed by a set of cylinders. A door (resp. gear) cylinder is locked when the door is closed (resp. extended or retracted). Of course, before starting moving, the cylinder, associated with the door/gear, should not be locked. So in the next refinement, we define two new variables:

- *door_cylinder_locked_p*
- *gear_cylinder_locked_p*

with the following invariant[3]:

INVARIANT
inv8: $door\_cylinder\_locked\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv9: $gear\_cylinder\_locked\_p \in \texttt{PositionsDG} \rightarrow \texttt{BOOL}$
inv10: $\forall po.(door\_cylinder\_locked\_p(po) = \texttt{TRUE} \Rightarrow door\_closed\_p(po) = \texttt{TRUE})$
inv11: $\forall po.(gear\_cylinder\_locked\_p(po) = \texttt{TRUE} \Rightarrow (gear\_ext\_p(po) = \texttt{TRUE} \vee gear\_ret\_p(po) = \texttt{TRUE}))$
inv12: $\forall po.(gear\_cylinder\_locked\_p(po) = \texttt{FALSE} \Rightarrow door\_open\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\})$

[3] The choice of the boolean type for variables *door_cylinder_locked_p* and *gear_cylinder_locked_p* are completly arbitrary. An other solution would be to define a new set $\{lock, unlock\}$.

In order to satisfy (**inv11**), we have refined the events *Start_GearExtend* and *Start_GearRetract* by adding the guard ($gear\_cylinder\_locked\_p(po) = \texttt{FALSE}$). Similarly, we have refined the event *Start_DoorClose* by adding the guard ($gear\_cylinder\_locked\_p = \texttt{PositionsDG} \times \{\texttt{TRUE}\}$) to make (**inv12**) satisfied. Finally, we have defined four new events to lock/unlock door/gear cylinders. In this paper, we provide only those associated with gear.

```
UnlockGearCylinder
  ANY po WHERE
    po ∈ PositionsDG
    gear_cylinder_locked_p(po) =TRUE
    (gear_ext_p(po) =TRUE ∨
                gear_ret_p(po)=TRUE)
    door_open_p =PositionsDG × {TRUE}
  THEN
    gear_cylinder_locked_p(po) :=FALSE
  END
```

and

```
LockGearCylinder
  ANY po WHERE
    po ∈ PositionsDG
    gear_cylinder_locked_p(po)=FALSE
    (gear_ext_p(po) =TRUE ∨
                gear_ret_p(po) =TRUE)
  THEN
    gear_cylinder_locked_p(po) :=TRUE
  END
```

4.4 Handle/switch/shock absorbers/hydraulic circuit modeling: the fifth refinement (machine HandSwitchShockAbsober

In this step, we continue the modeling of the monitored variables by introducing the handle, the analogical switch, the shock absorbers and the hydraulic circuit.

The behavior of the switch is described by Figure 4. Four states are distinguished:

- *Open*: initially, the switch is open and waits for the motion of the handle from Up to Down or inversely. The handle motion is represented in the figure by *handle?*.
- $Intermediate_1$: when the switch is open and the pilot moves the handle, the switch starts to close and its state becomes $Intermediate_1$. After a given amount of time (represented by the clock x in Figure 4), the switch becomes closed whatever the handle motion.

- *Closed*: when the switch is closed and in the absence of any handle motion, it starts to open after a given amount of time. In that case, the state of the switch moves to the state $Intermediate_2$.
- $Intermediate_2$: this state represents the switch which is opening. Two cases are to be distinguished: if the handle does not move, then the switch becomes completely open after a given amount of time, otherwise the switch starts closing by moving to the state $Intermediate_1$.

Let us recall that, for the moment, we do not take the timed aspects into account, that is, we consider that the change of states can occur at any time. The timed aspects will be deal with in Section 7.

First, we extend the context by defining two new sets `PositionsHandle` and `PositionsSwitch` to denote respectively the possible positions for the handle, *up* and *down*, and for the switch, *open*, *closed*:

CONTEXT
PositionsHandleSwitch
EXTENDS
PositionsDoorsGears
SETS
PositionsHandle
PositionsSwitch
CONSTANTS
up
down
open
closed
AXIOMS
partition(PositionsHandle, {*up*}, {*down*})
partition(PositionsSwitch, {*open*}, {*closed*})

Then, we define two variables to model the position of the handle and the switch respectively: *handle_p* and *analogical_switch_p*.

VARIABLES *handle_p*, *analogical_switch_p*
INVARIANTS
handle_p ∈ PositionsHandle
analogical_switch_p ∈PositionsSwitch

Since the analogical switch closes each time the handle changes its position, we add a Boolean variable *handle* which memorizes the handle shift. The events `PutHandleUp` and `PutHandleDown` representing the handle shift are as follows:

PutHandleUp
WHEN
handle_p = *down*
THEN
handle_p := *up*
handle :=TRUE
END

and

PutHandleDown
WHEN
handle_p = *up*
THEN
handle_p := *down*
handle :=TRUE
END

To model the physical behavior of the analogical switch depicted in Figure 6, we define two additional Boolean variables $Intermediate_1$ and $Intermediate_2$ that cannot be true at the same time as follows:

INVARIANT
inv13: ¬($Intermediate_1$ =TRUE ∧ $Intermediate_2$ =TRUE)
inv14: ($Intermediate_1$=TRUE ∨ $Intermediate_2$ =TRUE) ⇒
analogical_switch_p = *open*

Each transition is translated into an event whose guard corresponds to its source state and includes the condition (*handle*=TRUE) if it is triggered by the handle shift. The action of this event consists in assigning `FALSE` to the source state and `TRUE` to the target one. For the sake of space, we only provide the Event-B translation of two transitions.

close_Switch
WHEN
$Intermediate_1$ =TRUE
THEN
analogical_switch_p := *closed*
$Intermediate_1$:=FALSE
END

and

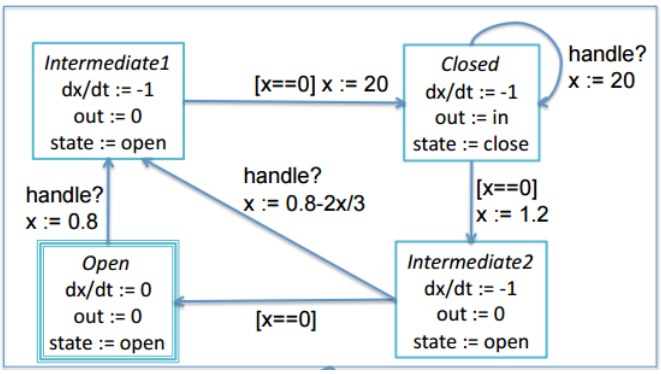


Fig. 4. Physical behavior of the analogical switch (Taken from [4])

HandleFromIntermediate2ToIntermediate1
WHEN
$Intermediate_2$ =TRUE
$handle$ =TRUE
THEN
$handle$:=FALSE
$Intermediate_2$:=FALSE
$Intermediate_1$:=TRUE
END

The hydraulic circuit is modeled with a Boolean variable $circuit\_pressurized\_p$ that states whether it is pressurized or not:

VARIABLES $circuit\_pressurized\_p$
INVARIANTS
$circuit\_pressurized\_p \in$ BOOL

and two events $Unpressurize$ and $Pressurize$ that change its state:

Unpressurize_HydrolicCircuit
WHEN
$circuit\_pressurized\_p$=TRUE
THEN
$circuit\_pressurized\_p$:= FALSE
END

and

Pressurize_HydrolicCircuit
WHEN
$circuit\_pressurized\_p$=FALSE
THEN
$circuit\_pressurized\_p$:=TRUE
END

In addition, we have refined each event related to the doors/gear/ motion and lock/unlock cylinders by adding a guard ($circuit\_pressurized\_p$ = TRUE). Finally, we model the gear shock absorbers by a Boolean variable that gives for each position the state of its associated shock absorber according to the following invariant stating that a gear shock absorber is on ground only if its gear is extended:

VARIABLES $gear\_shock\_absorber\_p$
INVARIANTS
$gear\_shock\_absorber\_p \in$ PositionsDG $\rightarrow$ BOOL
inv15: $\forall po.(po \in$ PositionsDG $\wedge$
$gear\_shock\_absorber\_p(po)$ =TRUE
$\Rightarrow$
$gear\_ext\_p(po)$=TRUE)

So, to preserve invariant (**inv15**), we refine the event Start_GearRetract by adding an action that set the variable $gear\_shock\_absorber\_p(po)$ to FALSE. In addition, to make the state of a shock absorber evolve, we have defined two new events: a first one to set it to FALSE and a second one to set it to TRUE under the guard that its gear is extended:

Make_GearAbsorberTrue
ANY
po
WHERE
$po \in$ PositionsDG
$gear\_shock\_absorber\_p(po)$=FALSE
$gear\_extended\_p(po)$=TRUE
THEN
$gear\_shock\_absorber\_p(po)$:=TRUE
END

and

```
Make_GearAbsorberFalse
  ANY
    po
  WHERE
    po ∈ PositionsDG
    gear_shock_absorber_p(po)=TRUE
  THEN
    gear_shock_absorber_p(po) := FALSE
  END
```

5 Modeling the controlled variables: the sixth refinement (machine *ValvesLignts*)

This section deals with the modeling of valves and lights that are controlled by the system. We describe how a valve becomes active/not active and how a light becomes on/off. Each valve is modeled with a Boolean variable (*general_EV_p*, *open_EV_p*, *close_EV_p*, etc.) to say whether it is active or not:

```
INVARIANTS
    general_EV_p ∈ BOOL
    open_EV_p ∈ BOOL
    close_EV_p ∈ BOOL
    extend_EV_p ∈ BOOL
    retract_EV_p ∈ BOOL
```

Also, two events are associated with each valve to make it active or not. In this paper, we describe the events that activate the open door valve and deactivate the extend valve; the others are very similar.

```
MakeOpenDoorValveActive
  WHEN
    open_EV_p= FALSE
    circuit_pressurized_p= TRUE
  THEN
    open_EV_p:= TRUE
  END
```

and

```
MakeExtendValveActive
  WHEN
    extend_EV_p= FALSE
    circuit_pressurized_p= TRUE
  THEN
    extend_EV_p:= TRUE
  END
```

In addition, we refine each event related to the motion of doors/gear by adding a guard to specify that the corresponding valve is active and its opposite is deactivated. For instance, we refine `Start_GearExtend` by adding the guard:

$$(extend\_EV\_p = \texttt{TRUE} \wedge retract\_EV\_p = \texttt{FALSE})$$

We also refine the events related to lock/unlock of door/gear cylinders by adding the adequate guard. For instance, we refine the event `LockGearCylinder` by adding the guard:

$$(gear\_ext\_p(po) = \texttt{TRUE} \wedge extend\_EV\_p = \texttt{TRUE}) \vee (gear\_ret\_p(po) = \texttt{TRUE} \wedge retract\_EV\_p = \texttt{TRUE})$$

Finally, the event `Pressurise_HydrolicCircuit` (resp. `Unpressurize_HydrolicCircuit`) is refined by adding the guard ($general\_EV\_p = \texttt{TRUE}$) (resp. $general\_EV\_p = \texttt{FALSE}$).

The lights are dealt with similarly to the valves. We model each of them by a Boolean variable (*greenLight_p*, *orangeLight_p*, *redLight_p*) and define two events for green and orange lights; one to set the light on and the other to set it off. For the red light, only the event that makes it on is defined since this state is kept forever:

```
MakeColorLightActive
  WHEN
    ColorLight_p=FALSE
  THEN
    ColorLight_p:=TRUE
  END
```

where $Color \in \{Green, Orange\}$ and

```
MakeColorLightNoActive
  WHEN
    ColorLight_p=TRUE
  THEN
    ColorLight_p:=FALSE
  END
```

where $Color \in \{Green, Orange, Red\}$.

6 Modeling the input/output variables and the controller: the seventh refinement (machine *Sensors*)

In this section, we describe how the controller takes its decisions about the setting of the lights and the activation/deactivation of the valves according to the information it gets from the sensors that it periodically reads. To do that, the controller reads the status of the handle,

the switch, the hydraulic circuit, the doors and the gear[4]. So, we introduce for each of these elements a new variable that represents its state as seen by the controller. Such variables are suffixed by "*_ind*" and are of the same type and have the same constraints as their associated variables suffixed by "*_p*". For instance, a door cannot be seen open and closed at the same time. The controller acquires information from the sensors as depicted in Table 1.

The key point of the event `ReadInput` is that each sensor does not give information that goes against the security of the system (the sensors are intrinsically safe), that means that if it says that a door/gear is {open, close}/{extended/retracted} then it is really the case. If the sensor is faulty, it should say: I do not know!, that is, it will return `FALSE` for the doors and the gear. From these inputs, the controller takes decisions about sending orders to the valves. Each order to a valve is modeled by a Boolean variable (*general_EV*, *close_EV*, etc.) such that:

INVARIANTS
inv16: $\neg(open\_EV = \texttt{TRUE} \wedge close\_EV = \texttt{TRUE})$ //Req R_{41}
inv17: $\neg(extend\_EV = \texttt{TRUE} \wedge ret\_EV = \texttt{TRUE})$ //Req R_{42}
inv18: $(open\_EV = \texttt{TRUE} \vee close\_EV = \texttt{TRUE}) \Rightarrow general\_EV = \texttt{TRUE}$
inv19: $(extend\_EV = \texttt{TRUE} \vee ret\_EV = \texttt{TRUE}) \Rightarrow open\_EV = \texttt{TRUE}$ //$inv18 + inv19 = R_{51}$

For instance, the controller sends orders to the general and extend valves as follows:

– when the analogical switch is closed, it sends a start stimulation to the general valve if it reads that the handle is `up` (resp. `down`) but the gear are not locked up (resp. down). It should also maintain the stimulation of the general valve if the open/close valve is still stimulated. The event that models the start/stop of the stimulation of the general valve is as follows:

[4] In this paper, we make the assumption that there is a unique sensor on each of these elements.

`OutputGeneralValve`
ANY *general_EV_value* **WHERE**
general_EV_value =**bool**(
(*analogical_switch_ind* =`closed` ∧
((*handle_ind*=`down` ∧
gear_ext_ind ≠`PositionsDG` × {`TRUE`})
∨
(*handle_ind*=`up` ∧
gear_ret_ind ≠ `PositionsDG` × {`TRUE`}))
)
∨ *open_EV* =`TRUE` ∨ *close_EV* =`TRUE`)
general_EV ≠ *general_EV_value*
THEN
general_EV := *general_EV_value*
END

– if the open door valve is stimulated and the doors are seen open, it sends a stimulation order to the extend valve if it sees that the handle is down but one of the gear is not extended and locked in the down position, otherwise it stops it:

`OutputExtendGearValve`
ANY *extend_EV_value* **WHERE**
extend_EV_value =**bool**(*handle_ind* =`down`
gear_ext_ind ≠ `PositionsDG`×{`TRUE`}
open_EV =`TRUE` ∧ *ret_EV* =`FALSE`
door_open_ind=`PositionsDG`×{`TRUE`})
extend_EV ≠ *extend_EV_value*
THEN
extend_EV := *extend_EV_value*
END

Similarly to the valves, the controller sends orders to the lights. At this level, we only introduce the order to the green and orange lights; the red one is achieved later when we model failures. For instance, when the controller sees the gear extended and locked, it sends order *gears_locked_down* as follows:

gears_locked_down :=**bool**(
gear_ext_sensor_valueF =`TRUE` ∧
gear_ext_sensor_valueL =`TRUE` ∧
gear_ext_sensor_valueR =`TRUE`
)

In this step, we refine each event that makes a valve active/not active by adding a guard to specify that its related order has been sent from the controller. For instance, we refine the event `MakeExtendValveActive` as follows:

```
ReadInput
  ANY
    handle_sensor_value , analogical_switch_sensor_value, circuit_pressurized_sensor_value,
    gear_ext_sensor_valueF, gear_ext_sensor_valueL, gear_ext_sensor_valueR,
              ...
  WHERE
    handle_sensor_value ∈ PositionsHandle
                    ...
    gear_ext_sensor_valueF ∈ BOOL ∧
          gear_ext_sensor_valueF=TRUE ⇒
        gear_ext(front)=TRUE ∧ gear_cylinder_locked_p(front)=TRUE
                    ...
  THEN
    handle_sensor_ind:=handle_sensor_value
                    ...
    gear_ext_sensor_p:= {front ↦ gear_ext_sensor_valueF,
        left ↦ gear_ext_sensor_valueL,right ↦ gear_ext_sensor_valueR}
                    ...
  END
```

Table 1. Reading inputs through the sensors

```
MakeExtendValveActive
  WHEN
    extend_EV_p= FALSE
    circuit_pressurized_p= TRUE
    extend_EV:= TRUE
  THEN
    extend_EV_p:= TRUE
  END
```

We also refine the events acting on the lights by adding a guard that expresses that the setting order has been received from the controller. For example, we refine the event `MakeGreenLightActive` into:

```
MakeGreenLightActive
  WHEN
    GreenLight_p=FALSE
    gears_locked_down=TRUE
  THEN
    GreenLight_p:=TRUE
  END
```

7 Introducing timing aspects: the eighth refinement (machine *TimedAspects*)

In this system, timing aspects are four folds: (1) the analogical switch takes time to move from open to close and vice versa (2) the start/stop stimulation of valves should be separated by some time, (3) the valves take time to be active, the cylinders take time to move and be locked/unlocked, (4) the controller has to read some inputs at given moments to be sure that the system behaves correctly as expected or not. In this section, we deal with the first three aspects and postpone the last one to the next section.

In the literature, there is a number of models that are very suitable for the modeling of the timing constraints like Alur and Dill's timed [5] or Dutertre and Sorea's calender automata [10]. We do not investigate these model since this paper aims at experiencing the Event-B method on the case study and drawing some conclusions on its applicability even if we know in advance that no discrete neither real time is directly supported in Event-B. However, some work have proposed to enrich the Event-B language by introducing time constraints [8,21]. In [8], the introduction of the timed aspects in Event B consists in defining a global clock *time* to count the elapsed time and a set of active timers *at* to store the next moments at which timed events should be executed. The execution of each event remove the value of the current time from the set *at*. We have explored this solution for the case study but it was unsuccessful since several events may have the same deadlines. In fact, the first to be executed will remove the deadline from the set *at* permitting at the same time the possible progression of the global clock and deadlock of the other timed events.

In [21], the authors proposed an Event-B modeling for three categories of discrete timing properties : *deadline, delay, expire*. These properties share a starting point that consists of the execution of an action A, and have the following semantics:

- *deadline* means that an event B must occur within a given time after the occurrence of the event A

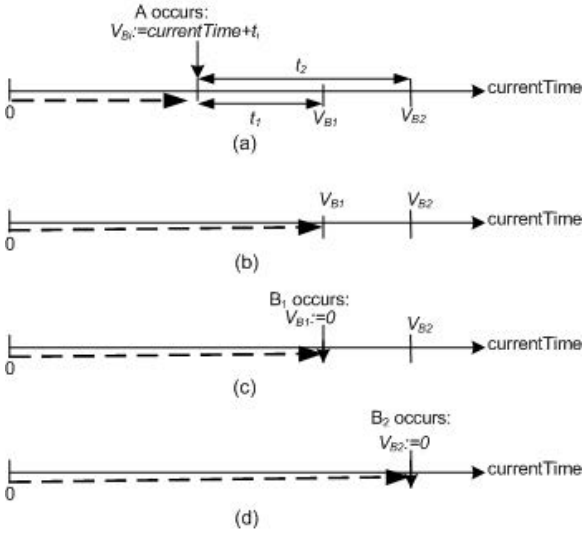


Fig. 5. Introduction of the time aspects in B: the principle

- *delay* means that an event B cannot occur within a given time after the occurrence of the event A
- *expire* means that an event B cannot occur after a given time after the occurrence of the event A

Since we are interested in events that take precise amounts of time to complete and others that must occur at defined moments, we use the *deadline* and *delay* categories but also a combination of both. Basically, we define a natural variable $currentTime$ to represent the current time of the global clock, and a timer variable v_B for each event B whose execution is time-constrained by the execution of an other event A. For instance, if we want the event B_i to be executed t_i $u.t$ (units of time) after the execution of the event A, then the execution of the event A has to set v_{B_i} to $(currentTime + t_i)$ (See Figure 5.a). The time continues to progress if there is at least one non-null timer ($v_{B_i} > 0$) by reaching the next deadline (v_{B_1} in Figure 5.b). Then, the time stops progressing until the associated event B_1 occurs. When B_1 occurs, the variable v_{B_1} is reset (See Figure 5.c), and the timer continues to progress until the next deadline (v_{B_2} in Figure 5.d). When the event B_2 occurs and the variable v_{B_2} is reset, the time stops progressing since there is no active timer. The process that represents the time progress is specified as follows:

```
passingTime
  ANY step WHERE
    step ∈ N1 ∧
    ⋁_i v_{B_i} ≠ 0∧
    ⋀_i(v_{B_i} ≠ 0 ⇒ currentTime + step ≤ v_{B_i})
  THEN
    currentTime := currentTime + step
  END
```

Finally, each timed event B_i includes in its precondition the guard : $(currentTime = v_{B_i})$.

7.1 Timing constraints on the analogical switch

As stated before, the current modeling of the behavior switch is achieved regardless to any timed aspects (See Section 4.4). Indeed, we considered that switch does not take time to change its state; which is not true. Indeed according to Figure 6, the switch takes 8 $u.t$ to move from the state $Open$ to the state $Intermediate_1$.

To introduce timing constraints on the switch, we add a natural variable *deadlineSwitch* that represents the deadline at which the switch changes its state. So, Figure 4 is transformed into Figure 6 where the timer x is replaced by the variable *deadlineSwitch* which is equal to $(currentTime + x)$. Such a transformation helps us to express some properties easier. So, to move from a state to another, the value of $currentTime$ should be equal to $deadlineSwitch$, then the deadline is updated adequately.

Event `HandleFromIntermediate2ToIntermediate1` is refined by adding the action:

$$deadlineSwitch := currentTime + (8 - (2/3) \times (deadlineSwitch - currentTime))^5$$

Similarly, the event `close_Switch` is refined by adding the following guard:

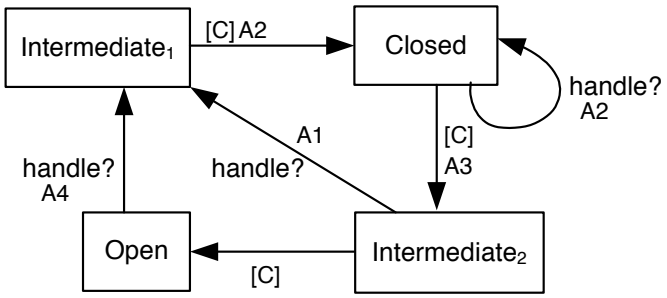
$$currentTime = deadlineSwitch$$

and action:

$$deadlineSwitch := currentTime + 200$$

7.2 Timing constraints on the start/stop stimulation of valves

In this system, the time between starting the stimulation of the general valve and the others should be separated by at least 2 $u.t$. Since the open valve is the first to stimulate just after the general valve, it is sufficient to respect this time between them. Similarly, the time between stopping the stimulation of the general valve and the others should be separated by at least 10 $u.t$. Since the open/close valve is the last valve that stops stimulation just before stopping the general valve, it is sufficient to respect this time only between them. In addition, the stimulation of contrary orders should be separated by at least 1 $u.t$. So, we have defined five natural variables:



C: currentTime =deadlineSwitch
A1: deadlineSwitch:=currentTime+(8-2/3*(deadlineSwitch-currentTime)
A2: deadlineSwitch:=currentTime+200
A3: deadlineSwitch:=currentTime+12
A4: deadlineSwitch:=currentTime+8

Fig. 6. Physical behavior of the analogical switch with deadlines

- $allowedStopGeneralEv$
- $allowedStartOpenEV$
- $allowedStartCloseOpenEV$
- $allowedStartExtEV$
- $allowedStartRetEV$

These variables are updated as follows:

- when the general (resp. open, close, extend, retract) valve is stimulated, then $allowedStartOpenEV$ and $allowedCloseOpenEV$ (resp. $allowedCloseOpenEV$, $allowedStartOpenEV$, $allowedStartRetEV$, etc.) are updated with the adequate value.
- when open (resp. close) valve is stopped then the variable $allowedStopGeneralEv$ is also updated with the adequate value.

So, the event `OutputGeneralValve` is refined as follows:

```
OutputGeneralValve
  ANY general_EV_value WHERE
    general_EV_value =bool(
    (analogical_switch_ind =closed ∧
      (
      (handle_ind=down ∧
        gear_ext_ind ≠PositionsDG × {TRUE})
      ∨
      (handle_ind=up ∧
        gear_ret_ind ≠ PositionsDG × {TRUE})
      )
    )
    ∨ open_EV =TRUE ∨ close_EV =TRUE)
    general_EV ≠ general_EV_value
    (currentTime ≥ allowedStopGeneralEv)
  THEN
    general_EV := general_EV_value
    allowedStartOpenEv:=
      {FALSE ↦ TRUE ↦ currentTime+2,
        TRUE ↦ FALSE ↦ 0}
          (general_EV ↦ general_EV_value)
    allowedStopGeneralEv :=0
  END
```

7.3 Timing constraints on the activation of the valves

A valve takes some time to be active/not active after starting/stopping its stimulation. So, we have associated with each kind of valves (general, door, gear) a natural variable that states the time at which the valve can be active/not active. For instance, we have defined the variable $deadlineStimulRetExtEv$ for the extend/retract valves. This variable is updated to the adequate time when the controller sends an order for these valves, and it is reset to 0 when the valve becomes active/not active. Basically, we have refined the event `OutputExtendGearValve` as follows:

```
OutputExtendGearValve
  ANY extend_EV_value WHERE
    extend_EV_value =bool(handle_ind =down
    gear_ext_ind ≠ PositionsDG×{TRUE}
    open_EV =TRUE ∧ ret_EV =FALSE
    door_open_ind=PositionsDG×{TRUE})
    extend_EV ≠ extend_EV_value
  THEN
    extend_EV := extend_EV_value
    deadlineStimulRetExtEv:=
  {FALSE ↦ TRUE ↦ currentTime+10,
    TRUE ↦ FALSE ↦ currentTime+36}
        (extend_EV ↦ extend_EV_value)
  END
```

Finally, we have refined the events that make the extend/retract valve active/not active by adding the guard ($currentTime = deadlineStimulRetExtEv$) and action ($deadlineStimulRetExtEv := 0$) to reset the deadline after executing the event.

7.4 Timing constraints on the cylinders

The gear/door cylinders take some time to lock/unlock but also to move from high to down and vice versa. To consider the time taken by a gear cylinder to lock/unlock, a variable $deadlineUnlockLockGearsCylinders$ has been defined. This variable is set by the events that make the extend/retract valve active in order to launch the deadline for unlocking the cylinders, and the events `Make_GearExtended` and `Make_GearRetracted` to launch the deadline for locking the cylinders. Similarly, a variable $deadlineGearsRetExt$ has been defined to consider the time taken by the gear to move from down to up and vice versa. So, the event `MakeExtendValveActive` is refined by adding the guard and actions that permit to reset the deadline and to set the moment at which the gear cylinders become unlocked.

```
MakeExtendValveActive
  WHEN
    extend_EV_p= FALSE
    circuit_pressurized_p= TRUE
    currentTime=deadlineStimulRetExtEv
  THEN
    extend_EV_p:= TRUE
    deadlineStimulRetExtEv:=0
    deadlineUnlockLockGearsCylinders:=
          PositionsDG×{currentTime+4}
  END
```

In addition, we refine the event `Start_GearExtend` by:

```
Start_GearExtend
  ANY po WHERE
    po ∈ PositionsDG
    gear_ret_p(po) =TRUE
    deadlineUnlockLockGearsCylinders(po)=0
  THEN
    gear_ret_p(po):=FALSE
    deadlineGearsRetExt(po):=
        {front ↦ currentTime+12,
          left ↦ currentTime+16,
            right ↦ currentTime+16}(po)
  END
```

8 Introducing failures: the ninth refinement (machine *Failures*)

8.1 Modeling failures

So far, we have considered all the physical elements as working correctly as expected. However in practice, each of them can fail: the switch, the cylinders and the valves can fail at any time. To take such failures into account, we have added for each of these elements an event that makes it fail. For example, for the switch and the door cylinders, we have defined the two following events where the variables $analogical\_switch\_fail$ and $door\_cylinder\_fail$ denote Boolean variables that say respectively whether the switch or a door cylinder has failed:

```
MakeSwitchFail
  WHEN analogical_switch_fail =FALSE
    analogical_switch_fail :=TRUE
  END
```

and

```
MakeDoorCylinderFail
  ANY po WHERE po ∈ PositionsDG THEN
    door_cylinder_fail(po) :=TRUE
  END
```

Consequently, we refine each event related to the behavior of the switch, the valves and the cylinders by adding a guard stating that the element change its status only if it has not failed. For instance, we have added the guard ($door\_cylinder\_fail(po)$:=`FALSE`) for the events `Make_DoorOpen`, `Start_DoorOpen`, `Start_DoorClose`, etc.

8.2 Detecting anomalies

As stated in the previous section, physical elements can fail. The controller does not have any information about that but it can deduce it by monitoring the status of the switch, the doors, and the gear. In fact, if the controller sends an order to stimulate the open valve but the doors are not seen open after a given time, then it can assert that a problem has happened (in at least one physical element) by displaying the *anomaly* information to the pilot. To this aim, the controller has to read, through the sensors, the status of these elements at well-defined times. For instance, the controller has to verify that the switch is closed 10 $u.t$ after the handle has changed its position otherwise an anomaly is detected. To model that, we add a Natural variable $nextInputOpenSwitch$ that memorizes the time at which the controller must not see the switch open. This variable is updated by the event `ReadInput` which we refine by adding the following action:

$$nextInputOpenSwitch := \{\texttt{FALSE} \mapsto nextInputOpenSwitch, \texttt{TRUE} \mapsto currentTime{+}10\}(\mathbf{bool}(handle\_ind \neq handle\_sensor\_value))$$

As for other deadlines, to avoid the starvation problem, we refine the event `passingTime` by adding a guard stating that if the variable $nextInputOpenSwitch$ is not null then the time can progress but without exceeding it. In addition, the event `ReadInput` resets the variable $nextInputOpenSwitch$ when this deadline is reached and the verification performed. So, we add the following actions to the event `ReadInput`:

$$\begin{array}{l} nextInputOpenSwitch := \\ \quad \{\texttt{FALSE} \mapsto nextInputOpenSwitch, \texttt{TRUE} \mapsto 0\} \\ \qquad (\mathbf{bool}(currentTime = nextInputOpenSwitch)) \\ anomaly := \\ \quad \mathbf{bool}(currentTime = nextInputOpenSwitch \wedge \\ \qquad\qquad analogical\_switch\_ind = open) \end{array}$$

The other anomalies on the doors, the gear, the hydraulic circuit are dealt with similarly. In addition, we have refined the event `ReadInput` and the events sending orders to the valves by adding the guard ($anomaly$ =`FALSE`) in order to stop the system. Indeed according to the description of the system, the anomaly message has to be maintained forever. From a modeling point of view, we introduce a deadlock such that no operation becomes possible.

9 Verification of the development and the requirements (machine $PropertyVerification$)

This section describes the verifications and the verifications carried out using the integrated prover of Rodin but also using the model checker ProB and the animator AnimB[6] plug-ins with which we have extended the Rodin platform. Our strategy to verify the development and the different requirements is as follows. We used ProB and AnimB in the first refinement steps mainly to discover some errors prior to a proof phase that may be long and complex but also to verify some properties that need a big effort to be proved. More details are given in the following subsections.

9.1 Verification using ProB

We used ProB mainly to ensure that the invariant is not trivially falsified. When ProB finds a counter-example for the invariant, it provides the sequence of the operations that leads to an invalid state from the initial state. In such cases, we reworked and fixed our Event-B specification by adding/modifying some guards/actions. For example, we used ProB to verify the properties R_{41}, R_{42} and R_{51} that are specified as invariants of the Event-B model (see **inv16, 17, 18, 19**, page 11):

R_{41}. When the command line is working (normal mode), opening and closure doors electrovalves are not stimulated simultaneously.

$$\neg(open\_EV{=}\texttt{TRUE} \wedge close\_EV{=}\texttt{TRUE})$$

R_{42}. When the command line is working (normal mode), outgoing and retraction gear electrovalves are not stimulated simultaneously.

$$\neg(extend\_EV{=}\texttt{TRUE} \wedge ret\_EV{=}\texttt{TRUE})$$

R_{51}. When the command line is working (normal mode), it is not possible to stimulate the maneuvering electro-valve (opening, closure, outgoing or retraction) without stimulating the general electro-valve.

$$(extend\_EV = \texttt{TRUE} \vee ret\_EV = \texttt{TRUE}) \Rightarrow general\_EV = \texttt{TRUE}$$

[6] `http://www.animb.org/`

When ProB found no counter-example on the whole Event-B models; this gave us some confidence about the correctness of the specification before performing the proof activity. Finally as expected ProB detected a deadlock on the Event-B model specifying failures since we have made a choice to make the system stop when an anomaly happens.

9.2 Verification using AnimB

AnimB is an animator plug-in that permits to play different scenarios and check the behavior of the Event-B models with respect to the desired system by showing at each step the values of each variable, which events are enabled and which are not. For example, we animated the complete outgoing and retraction sequence by letting the handle in a same position during the whole sequence. Moreover, we checked the different possibilities by interrupting both sequences at each moment by a counter order of the handle. We used AnimB to verify requirements R_{11} and R_{12}:

R_{11}. When the command line is working (normal mode), if the landing gear command handle has been pushed DOWN and stays DOWN, then the gear will be locked down and the doors will be seen closed less than 15 seconds after the handle has been pushed.

R_{12}. When the command line is working (normal mode), if the landing gear command handle has been pushed UP and stays UP, then the gear will be locked retracted and the doors will be seen closed in less than 15 seconds.

Indeed, these dynamic properties relate two states (the present and the future) separated by several intermediate states. Verifying such properties using a proof or model-checking based approach would need a big effort in modeling them as invariants or LTL properties. We have especially tried to prove property R_{11} using the prover of Rodin, but we have not succeeded since several actions may be executed between the instant when the handle is pushed and the instant when the gear are locked down. So, several intermediate lemmas should be defined to characterize all these actions.

This tool also helped us when we have introduced the timed aspects by experimenting the approach described in [8]. As said before, the obtained results showed that it was not adapted for the case study as several events can have the same deadline.

Let us remark that AnimB was more helpful on the initial model and the first refinements since the number of events is low, but unfortunately it became less efficient for the last refinement that contains several variables and events; in that case the tool worked very slowly due to memory shortage.

9.3 Verification of other requirements (machine PropertyVerification)

Most requirements to verify are temporal properties that refer to several moments of the system. Moreover, these requirements (except requirements R_{41}, R_{42} and R_{51}) do not need to be specified as invariants since they can be deduced from the behavior of the system itself. That means that they have been already integrated during the development of the system through more elementary requirements. For instance, we know that the gear will be open in less 15 after the handle has been pushed because the sum of durations taken by the doors to open and the cylinders to unlock and move is less than 15. Consequently, to distinguish the specification of the system from the verification of such properties, we have created a new refinement level that defines such properties as invariants. This paper illustrates the verification of the properties through two examples. Table 2 gives the results of the verification activities. The verification of the other properties can be found at [1].

R_{74}. If one of the three gears is not seen locked in the down position more than 10 seconds after stimulating the outgoing electro-valve, then the Boolean output normal mode is set to false.

To specify this property, we have defined a new variable $TimeStimulExtRetEv$ to memorize the time at which the extend/retract valve is stimulated. This variable is set by the event `OutputExtendGearValve` by adding the action ($TimeStimulExtRetEv := currentTime$). Then, the property is specified as follows:

$$\left(\begin{array}{c} currentTime > TimeStimulExtRetEv + 100 \\ \wedge \\ extend\_EV = \texttt{TRUE} \end{array}\right) \Rightarrow \left(\begin{array}{c} anomaly = \texttt{TRUE} \\ \vee \\ gear\_ext\_ind = PositionsDG \times \{\texttt{TRUE}\} \end{array}\right)$$

To discharge this invariant, the following intermediate lemmas have been added:

$$\left(\begin{array}{c} currentTime > TimeStimulExtRetEv + 100 \\ \wedge \\ extend\_EV = \texttt{TRUE} \end{array}\right) \Rightarrow nextInputGearEndExtRet = 0$$

and

$$\left(\begin{array}{c} nextInputGearEndExtRet = 0 \\ \wedge \\ extend\_EV = \texttt{TRUE} \end{array}\right) \Rightarrow \left(\begin{array}{c} anomaly = \texttt{TRUE} \\ \vee \\ gear\_ext\_ind = \texttt{PositionsDG} \times \{\texttt{TRUE}\} \end{array}\right)$$

The first invariant ensures that the time does not progress beyond the deadline ($TimeStimulExtRetEv + 100$) without reading the state of the gear. Indeed, the variable *nextInputGearEndExtRet* is reset when the gear is read. The second one states that the controller sets the variables *anomaly* and *gear_ext_ind* correctly when the deadline is reached.

R_{21}. When the command line is working (normal mode), if the landing gear command handle remains in the DOWN position, then retraction sequence is not observed.

To specify this property, we have defined a new variable *stayDown* which is equal to TRUE when the handle stays in down position more than two controller cycles. A controller cycle denotes the following sequence of actions: *Reading inputs*, *processing inputs* and *sending outputs*. The variable *stayDown* is initialized to TRUE and updated each time the controller reads the position of the handle as follows:

$$stayDown := \textbf{bool}\left(\begin{array}{c} handle\_ind = down \\ \wedge \\ handle\_sensor\_value = down \end{array}\right)$$

Moreover, we have defined the variable *end_cycle* to know if we are at the end of the cycle. This variable is initialized to TRUE and updated as follows:

- it is set to FALSE when the controller is reading the different inputs
- it is set to TRUE when the controller sends the different outputs

Then, the property is specified as follows:

$$\left(\begin{array}{c} stayDown = \texttt{TRUE} \\ \wedge \\ end\_cycle = \texttt{TRUE} \end{array}\right) \Rightarrow retract\_EV = \texttt{FALSE}$$

The last formula states that the retract order is not sent by the controller if it sees the handle in the down position. To be discharged, we have added the following lemma that we have demonstrated in its turn:

$$handle\_ind = up \Rightarrow stayDown{=}\texttt{FALSE}$$

This lemma checks whether the variable *stayDown* is updated correctly: if the controller sees the handle in the up position, then the variable *stayDown* should be equal to FALSE.

10 Discussion and Conclusion

In this paper, we have presented a modeling of a landing gear system with the formal Event-B language. To this aim, we have proceeded into 3 main phases: (1) modeling the system without timed concerns and possible failures; (2) taking timed concerns into account; (3) considering the possible faults on the different elements of the system. Let us draw up some lessons learned from this case study and a summary of the key findings and conclusions.

10.1 From a technical point of view

We have defined 66 variables and 48 events split into 10 refinement levels that give rise to 285 proof obligations, 72% of which have been discharged automatically; we have accomplished the remaining proofs interactively thanks to the Atelier B and SMT provers which are Rodin plugins. It is undeniable that the use of these tools helped us not only during the verification step but also in the development of the modeling. The refinement concept of Event-B allows us to deal with the complexity of the system even if due to memory shortage the tool worked very slowly on the last refinement components.

Regarding the description of the case study, there are two requirements that we have not taken into account. First, we made the assumption that each sensor is unique and not triplicated. This is not a strong assumption and does not affect the modeling; it can be easily relaxed by only adapting the event ReadInput. For the handle for instance, we can define two functions *handle_sensors* and *handle_sensors_valid* to memorize the values of the sensors and its validity:

$$handle\_sensors \in 1..3 \longrightarrow \texttt{BOOL} \wedge handle\_sensors\_valid \in 1..3 \longrightarrow \texttt{BOOL}$$

Then, the event ReadInput is updated as depicted in Table 3 (value TRUE (resp. FALSE) represents position up (resp. down)).

Moreover, we only considered a unique piece of controller because the description of the case study is not clear about how the two modules can deliver different outputs while, from the design point of view, they work similarly and read the same inputs.

Requirement	Verified?	Method	Comment
R_{11},R_{12}	✓	Animation	Proof seems to be too hard since it needs several intermediate lemmas.
R_{21}	✓	Proof	It is verified from the instant where the controller sees the position of the handle down
R_{22}	✓	Proof	It is verified from the instant where the controller sees the position of the handle up
R_{31}	✓	Proof	It is not valid on the physical elements since the controller can start extending/retracting the gear when the doors are actually open but the close valve does not stop completely. Thus, we express it according to the internal variables.
R_{32}	✓	Proof	It is not valid on the physical elements since the controller can start opening/closing the doors when the gear are actually extended/ retracted but the extend/retract valve does not stop completely. Thus, we express it according to the internal variables.
R_{41}, R_{42}, R_{51}	✓	Proof and model checking	
R_{61}, R_{62}, R_{63}, R_{64}	✓	Proof	
R_{71}, R_{72}, R_{73}, R_{74}	✓	Proof	

Table 2. Verification results

ANY
handle_ind_value,
handle_sensor_valid_value
WHERE
$handle\_ind\_value =$
$((card(handle\_sensor\_valid^{-1}[\{\texttt{TRUE}\}])=3 \wedge$
$((handle\_sensor(1) = \texttt{TRUE} \wedge (handle\_sensor(2) = \texttt{TRUE} \vee$
$handle\_sensor(3) = \texttt{TRUE})) \vee$
$\vee$ (handle_sensor(2)=TRUE $\wedge$ handle_sensor(3)=TRUE)))
$\vee \quad (card(handle\_sensor\_valid^{-1}[\{\texttt{TRUE}\}])=2 \wedge$
$card(handle\_sensor[handle\_sensor\_valid^{-1}[\{\texttt{TRUE}\}]])=1)))$
$handle\_sensor\_valid\_value = \ldots$
THEN
$handle\_ind := \{\texttt{TRUE} \mapsto \texttt{up}, \texttt{FALSE} \mapsto \texttt{down}\}(handle\_ind\_value)$
$handle\_sensor\_valid := handle\_sensor\_valid\_value$
$\ldots$

Table 3. The modeling of the sensor triplication

10.2 From a methodological point of view

The main difficulty was to define a method to tackle the complexity of the case study. Indeed, the whole system can be viewed as tow subsystems (the controller and the environment) that interact with each other by exchanging information; and the main problems, in that case, were to identify the borders between these subsystems and to define a design approach to construct the formal model.

The combination of the four-variable model of Parnas and of the Event-B refinement process has proved very relevant for this kind of systems. The former allows to classify the variables that represent the system and its environment and the latter allows to gradually introducing these variables. This approach has already been proposed by several authors such as Hudon and Hoang [13] and Butler [6]. In the first paper, neither time aspects nor failures are taken into account. In [7], a very simple case study is presented. Contrary to this work, we have chosen to consider time constraints later in the design, since it seemed to us simpler for the proof activity. Finally, failures have been introduced at the end of the process following the idea of considering first the nominal system behavior as advised by [15,19].

Moreover we think that the modeling can be improved if Event-B and the Rodin framework, under which this development has been achieved, offer real-time aspects.

It would also be interesting to look deeper into the use of one of the structuring mechanisms proposed for Event-B: decomposition [22] or modularization [14], in order to structure the specification into logical units.

Finally, we will investigate the use of the ASTD formal graphical language [11,12] (Algebraic State Transition Diagrams), developed by the team of Marc Frappier from the Sherbrooke university, to model such complex systems. Indeed as noted before, the use of Event-B to express the dynamic properties, like ordering or liveness, requires some efforts; we think that the graphical notations could make this task easier.

Dedication

In loving memory of my late beloved mother, Fariza-Ourida Mokrani (died on January 26^{the} 2015). I can not thank you enough for the sacrifices you've made for my education and my well being. *May your rest be sweet as your heart was good*. Your Daughter Amel.

References

1. `http://deploy-eprints.ecs.soton.ac.uk/467/`
2. J-R. Abrial: The B-book, Assigning Programs to Meanings. Cambridge University Press, (2005)
3. J-R. Abrial: Modeling in Event-B - System and Software Engineering. Cambridge University Press, (2010)
4. F. Boniol and V. Wiels: The Landing Gear System Case Study, ABZ Case Study, Communications in Computer Information Science, volume 433, Springer, (2014)
5. R. Alur and D-L. Dill: A Theory of Timed Automata. Theoretical Computer Science, 126(2): 183-235 (1994)
6. M. Butler: Towards a Cookbook for Modelling and Refinement of Control Problems. Working Paper. ECS, University of Southampton, `http://deploy-eprints.ecs.soton.ac.uk/108/1/cookbook.pdf`, (2009)
7. M. Butler: Using Event-B Refinement to Verify a Control Strategy, Working Paper. ECS, University of Southampton, `http://deploy-eprints.ecs.soton.ac.uk/107/` (2009)
8. D. Cansell, D. Méry and J. Rehm: Time Constraint Patterns for Event B Development, Proceeding of 7th International Conference of B Users (B2007), 140–154 (2007)
9. E-M. Clarke and E-A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Logic of Programs, 52-71 (1981)
10. B. Dutertre and M. Sorea: Modeling and Verification of a Fault-Tolerant Real-Time Startup Protocol Using Calendar Automata, FORMATS/FTRTF, 199-214 (2004)
11. M. Frappier, F. Gervais, R. Laleau, B. Fraikin and R. St.-Denis: Extending Statecharts with Process Algebra Operators. ISSE 4(3): 285-292 (2008)
12. M. Frappier, F. Gervais, R. Laleau and J. Milhau: Refinement Patterns for ASTDs. Formal Aspects of Computing 26(5): 919-941 (2014)
13. S. Hudon and T.S. Hoang: Development of Control Systems Guided by Models of their Environment. Electronic Notes in Theoretical Computer Science, Volume 280, pages: 57-68 (2011)
14. A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic and T. Latvala: Supporting Reuse in Event-B Development: Modularisation Approach. ABZ'2010, Springer, LNCS 5977: 174-188, (2010)
15. R-D. Jeffords, C-L. Heitmeyer, M. Archer and E-I. Leonard: Model-Based Construction and Verification of Critical Systems using Composition and Partial Refinement, Formal Methods in System Design 37(2-3): 265-294 (2010)
16. M. Leuschel and M. Butler. Prob: A Model Checker for B. In FME 2003: Formal Methods, International Symposium of Formal Methods Europe, 855-874 (2003).
17. M. Leuschel and M.-J. Butler. ProB: An Automated Analysis Toolset for the B Method. in International Journal on Software Tools for Technology Transfer 10(2): 185-203 (2008)
18. D. Lorge Parnas and J. Madey: Functional Documents for Computer Systems: Science of Computer Programming 25(1): 41-61 (1995)
19. S-P. Miller and A-C. Tribble: Extending the Four-Variable Model to Bridge the System-Software Gap, Proceedings of the 20th Digital Avionics Systems Conference (DASC01), Daytona Beach, Florida (2001)
20. A. Pnueli: The Temporal Logic of Programs. In 18th Annual Symposium on Foundations of Computer Science, 46-57, (1977)
21. M-R. Sarshogh and M. Butler: Specification and Refinement of Discrete Timing Properties in Event-B, in Electronic Communication of the European Association of Software Science and Technology, Volume 46 (2011)
22. R. Silva, C. Pascal, T-S Hoang and M. Butler: Decomposition tool for Event-B. Software - Practice and Experience 41(2): 199-208 (2011)